The OddJob (OJ) Script Language Interpreter

for the OS-9/68000 and CD/RTOS operating systems

TECHNOTEACHER INCORPORATED

TTI

ACKNOWLEDGEMENTS

I would like to thank Victoria T. Newcomb for her typesetting and editorial contributions, and for being understanding about the many revisions this manual has gone through. I would also like to thank Anders Franzen for his proofreading and editorial contributions. Finally, I would like to thank the staff and students at the Florida State University Center for Music Research, and particularly Dr. Peter Spencer, for serving very competently as the test site for OJ.

PUBLICATION EDITOR: Steven R. Newcomb

REVISION: C

PUBLICATION DATE: December, 1990

PRODUCT NUMBER: 0101

TechnoTeacher Incorporated
1810 High Road
Tallahassee, Florida 32303-4408 USA
Telephone: 904 422 3574
Fax: 904 [illegible] 2562

Table of Contents

This page deliberately left blank.

Introduction to OddJob

OddJob (OJ) is an interpreter for programs written in the OJ script language; this manual describes the OJ language and the features of the OJ interpreter. The purpose of OJ is to make life easier for users of the OS-9/68000 and CD-RTOS operating systems.

You are, no doubt, already familiar with the idea of a "shell script," which is simply a list of program invocations (tasks) for the computer to do, in the order in which they appear in the script, together with certain directives to the shell program itself. When the shell program interprets a "shell script," it simply reads the shell script file line by line, and executes what is written on each line just as if it had been typed at the shell prompt. (If you are not familiar with the shell or with the concept of a shell script, it would be a good idea for you to read about the shell program in your OS-9/68000 operating system manual.) If you are familiar with the MS-DOS world, it might help you to know that a shell script is very much like a .BAT file in DOS.

OJ is not currently configured as a shell; it does not offer an interactive shell prompt to the user. Instead, it is a programming language which greatly expands the number and complexity of tasks possible to have accomplished automatically under the OS-9 operating system. It also offers special facilities which no DOS programming language can offer, which take advantage of OS-9's real-time multitasking and true piping capabilities. OJ perhaps most closely resembles the Unix C-Shell script language, but it includes within itself several of the features of several other Unix utilities, such as chat and awk. OJ's handling of variables and its string-processing facilities are especially awk-like, with its arbitrarily associative variable lists, `sprintf`, `fprintf`, and `substring` commands, and its `index()` function. Programmers who are already familiar with both the C language and awk will find OJ particularly easy to learn.

FLOW OF CONTROL

The OJ interpreter supports very genéralized structured flow-of-control commands. These include `if`, `else`, `endif`, `switch`, `case`, `default`, `break`, `loop`, and `continue`, which work very much like their counterparts in the C language. It also supports an unstructured `goto` command. Subroutine calls and returns can be made with the `include`, argumented `do`, and `return` commands.

STRING MANIPULATIONS

The OJ string manipulation command set includes `set` (which offers several string processing options), `arraytok`, `replace`, `alphameric`, `substring`, and `sort`.

CALCULATIONS, VARIABLES AND FUNCTIONS

OJ supports double-precision floating-point calculations, n-dimensional arrays of variables, and the same set of arithmetic and logical operators as is found in the C language. Special functions available within calculational expressions include `index()`, `namecmp()`, `strcmp()`, `length()`, `rand()`, `charval()`, `min()`, `max()`, `devrdy()`, `oct2dec()`, `hex2dec()`, `frac()`, `int()`, `isdir()`, `isfile()`, `isreal()`, `lvarnum()`, and `gvarnum()`. There are two classes of user variables: local (to the current script) and global. Random numbers, arrays of repeatable and non-repeatable random numbers, and automatic and manual random-number seeding are provided. Convenient means are provided (`vsave` and `vrestore`) to save and restore the contents of variables to and from named disk files.

There are several system-reserved pseudo-variables and arrays of pseudo-variables: `exitstat`, `argc`, `e_banner`, `pwd`, `systime`, `systimef`, `procid`, `userid`, `sendbuf[]`, `reply[]`, `ereply[]`, `qreturn`, `stderr`, `stdout`, `argv[]`, `curscript`, `LV_NAME[]`, `GV_NAME[]`, `LV_CONT[]`, `GV_CONT[]`, `LVARS`, and `GVARS`, and pseudo-constants used for identifying and interpreting error conditions: `E_...`, `e_msg[]`.

INPUT/OUTPUT

There is a convenient (and very C-like) set of input/output commands, including `fprintf`, `openfile`, `closefile`, `readfile`, `writefile`, `seekfile`, and `tellfile`. To write to the console, `fprintf $stdout` or `fprintf $stderr` are normally used.

SOCKET INTERACTIONS

With its unique "socket-interaction" command set, OJ is able to support scripts which are intended to simulate the behavior of an interactive user of one or more simultaneously-active processes. Such processes can be running both on the local computer and on remote computers. For example, a script can be used to operate a local interactive process which would otherwise require tedious or repetitive attention from a human operator, and it can be written in such a way as to occasionally consult another process (e.g., an interactive Prolog environ-

ment) running locally or remotely. The socket interaction command set includes set_waits, set_send_delay, ignore, send, socket, set_mbf (set match buffer), password, show_mbfs (show match buffers), strip_msb_on, and strip_msb_off. For debugging socket interactions, verbose_on and verbose_off commands are provided.

REDIRECTION TO AND FROM OJ VARIABLES

If the interactive control offered by the socket interaction commands is not needed, the redirect command allows processes to be launched in such a way that their standard input and output streams are directed from and/or to OJ variables.

DEBUGGING AND OPTIMIZATION FACILITIES

There is an interactive script debugger, with commands for setting and killing breakpoints, listing files, inspecting and setting variables, displaying status, etc.

In addition to the debugger, there is a debugging command set which includes debug_on(_off) , show_vars , show_labels , echo_on(_off) , show_tokens_on(_off) , show_comlines_on(_off) , and sys_exec_on(_off).

Elapsed time measurement commands include marktime, difftime, and julian.

Other miscellaneous commands include chd, chx, set_prior(ity), and wait.

SYSTEM REQUIREMENTS

As supplied for use with MC68000-based systems, the OJ executable module is about 85 kilobytes in size. Each OJ process requires about 12 kilobytes of RAM in order to be launched; the amount of memory used after that depends entirely on how many variables and labels are used, and how much data each variable must accommodate. There is no built-in limit on the number or size of variables that OJ can support. (The memory allocated to a variable can be recovered for re-use by OJ via the unset command.)

This page deliberately left blank.

Installation

Any last-minute notes about installation and operation of OJ are in the README file on your OJ disk. Read it first.

Your installation disk has an OJ script on it which will assist you in installing OJ on your OS-9/68000 computer system. It's easy to do; merely execute the installation script and answer the questions it asks you interactively.

To execute the installation script:

(STEP 1) Log into your system as the super-user (user number 0.0).

(STEP 2) Insert the installation disk into your *floppy disk device* (e.g., /d0).

(STEP 3) Change your current directory to the root directory of that *floppy disk device*:

```
OS-9: chd /d0
```

(STEP 4) Load oj:

```
OS-9: load -d oj
```

(STEP 5) Execute the install script:

```
OS-9: oj install
```

From here on, you will be asked several questions about where you want various things to be put on your hard disk, etc. When these questions have been answered, the install script will handle the tedium of installing the program.

The install script will copy the OJ program itself to the directory you specified, create a default directory for OJ scripts, copy some sample scripts to that directory, edit your .login file so as to set the OJ_PATH environment variable, etc.

Alternatively, if you do not choose to use the automatic installation script, you may install OJ yourself as follows:

(1) Copy OJ to a directory of executables. Having copied OJ to its new directory, make sure OJ's execute permissions are set correctly, and make sure your .login file (see the OS-9 manual regarding the /dd/SYS/password file) sets the PATH shell environment variable in such a way that OS-9 will look in that directory for OJ when it is invoked. (Here at Technoteacher, Inc., we always place executables not supplied with the OS-9/68000 operating system in a directory called /dd/USR/CMDS. In fact, we put everything not supplied by Microware in /dd/USR, because that enables us to update the entire operating system without first having to extract from the old version and re-insert into the new version all the files that were not supplied by Microware.)

(2) Copy all the scripts from the */floppy disk device*/SCRIPTS directory to a directory on your hard disk which you have set aside for such scripts. At Technoteacher, Inc., we use /dd/USR/SCRIPTS for this purpose.

(3) Copy the contents of the */floppy disk device*/DOC directory to your documentation directory. At Technoteacher, we use /dd/USR/DOC for this purpose. If disk space is in short supply, you may wish to copy only the ojdb.doc file, which is automatically read when the ? debugger directive is issued.

(4) Edit your .login file in such a way that the OJ_PATH shell environment variable is set to the full pathnames of the directories in which you wish OJ to look for executable scripts. Presumably, one of these will be the one you used in step 2, above. See Section 3 of this manual for more information about OJ_PATH. If the OJ_PATH shell environment variable has not been set, OJ will look first in the current data directory, and then in /dd/USR/SCRIPTS; it is just as if there were the following line in your .login file:

```
setenv OJ_PATH .:/dd/USR/SCRIPTS
```

(5) Edit your .login file in such a way that the OJ_DOC shell environment variable is set to the full pathname of the directory where you put all the OJ documentation. For example, you might insert the following line:

```
setenv OJ_DOC /dd/USR/DOC
```

However, if you put the documentation in /dd/USR/DOC, it is unnecessary to set the OJ_DOC environment variable at all. OJ will look there by default if the OJ_DOC shell environment variable has not been set.

Invocation of OJ

Normally, one invokes OJ as follows:

```
OS-9: oj (opts) name_of_script (args_to_script) (opts)
```

Options governing the operation of OJ all begin with two hyphens (--); this allows OJ scripts themselves to have options that begin with a single hyphen. However, like most OS-9 utilities,

```
OS-9: oj -?
```

will cause a usage statement to be displayed. OJ's options are:

`-?` Show an OJ invocation line synopsis.

`--b` Specify the size (in bytes) of the buffer to be used for reading the script itself and all scripts `done` or `included` by the script. The minimum buffer size is 512 bytes. The purpose of the `--b` option is to allow you to minimize memory usage when running large and/or highly recursive scripts, at some sacrifice in execution speed. In the absence of a `--b` option, each script will get a buffer just big enough to accommodate the entire script; this reduces execution time by minimizing disk accesses, especially during iterations of large `loops`.

`--d` Operate in debug mode. See Section 11.

`--e` Force echoing of each executable line to the console, overriding any `echo_off`. (See the discussion of the `echo_on` command.)

`--q` Prohibit verbose output, overriding any `verbose_on`. (See the discussion of `socket`, `send`, and related commands.)

`--v` Force verbose output, overriding any `verbose_off`. (See the discussion of `socket`, `send`, and related commands.) Also affects the `wait` command.

`--x` Prohibit system command execution, overriding any `sys_exec_on`. (This prevents all separate programs normally invoked by the script from actually being invoked.)

OJ may be instructed to look for scripts in any directories. Use the "setenv" command at an OS-9 shell prompt (it is best to do this in your /dd/startup or your own .login file):

```
OS-9: setenv
OJ_PATH.:/h0/usr/scripts:/h0/usr/srn/scripts
```

The above setting of the OJ_PATH shell environment variable causes OJ to look first in the current working (i.e., data) directory for scripts, then in /h0/usr/scripts, and finally in /h0/usr/srn/scripts.

If you don't do a "setenv OJ_PATH ... ," OJ will look first in the current working directory, and then in /dd/usr/scripts. Of course, if you provide the full pathlist of the script, beginning with a slash and a device name, that will be the only script file OJ will attempt to open.

Alternatively, you may set a local and/or a global variable within an OJ script, which will override the effect of any environment variable you set with setenv at an OS-9 prompt:

```
# inside a script now
set OJ_PATH =.:/h0/usr/scripts:/h0/usr/srn/scripts
```

The OJ interpreter executes actual command lines as they appear in a given .oj script. There is exactly one command at the beginning of each line.

For each line, the interpreter does the following four things in order:

(1) tokenizes the command line;

(2) performs macro substitution;

(3) performs variable substitution;

(4) executes the command.

STEP 1: TOKENIZATION OF THE COMMAND LINE

The command line is scanned for its "tokens," the smallest units meaningful to the interpreter. Tokens are separated by whitespace (i.e. any number of spaces and/or tabs). To include whitespace characters in a token, surround the entire token with double quotation marks. Double quotation marks which surround tokens are discarded. To include a double quotation mark within a token, precede it with a backslash (i.e., \").

```
# example showing tokenization
token0 token1 "This is all token2"
token0 "This \"token\" contains double quotation
marks"
```

STEP 2: MACRO SUBSTITUTIONS

The macro substitution feature of OJ provides the only way to create multi-token arguments with a single symbol. To set up a macro, first set a variable to the string you don't want to write repeatedly:

```
# set the variable "my_error"
sprintf my_error \
   "fprintf $stderr \"this program aborted because %%s\n\""
```

Now, `$my_error` will contain:

```
fprintf $stderr "this program aborted because %s\n"
```

Thereafter, the command line:

```
$(my_error) "I blew it"
```

will (before it is executed) automatically become:

```
fprintf $stderr "this program aborted because %s\n" "I blew it"
```

Note the use of parentheses in "`$(my_error)`." The parentheses indicate to the interpreter that macro substitution must be done. Note that in the example above, a single token, `$(my_error)`, became three tokens as a result of macro substitution.

Multi-line macros are not supported. The entire macro must take place on one and only one line. (It will not do any good to use a \ character in your efforts to make a longer line.) Neither recursive macros nor argumented macros are supported.

STEP 3: VARIABLE SUBSTITUTION (see also Section 6)

Tokens that begin with $ are assumed to consist entirely of a $ immediately followed by the name of a variable to which a value has already been assigned. (Any command that assigns a value to a variable will create that variable if it does not already exist.) In this third step, whenever encountered, such a token is replaced by the content of the variable whose name follows the $.

A variable name may be any length, and must begin with an alphabet character of either upper or lower case; thereafter it may contain any combination of the following characters only: any alphabet characters of either case, any numeric characters (0 through 9), periods (.), underscores (_), and left and right square brackets ([and]).

The square brackets are treated as array index operators (see Section 6 for a discussion of arrays).

The contents of all variables are actually strings, even if their content is numeric. If the variable name given after a $ does not exist, OJ will abort (or, if OJ was invoked with the `--d` option, give the user an `OJdb:` [interactive debugger] prompt); therefore, you must always be sure to set a variable to some (any) value prior to the time OJ encounters it with a $ before it. If the variable does exist, the content of that variable (a string) will be substituted for the token.

Because certain commands must perform character substitutions (see Section 9) their variable

substitutions are not done in this step 3; rather, they are done later during execution of the command-specific software in the interpreter. Generally this doesn't make any difference, except that it does give you a way to avoid variable substitution altogether: if you are using a backslash-interpreting command and you must begin a token with $, precede the $ with a backslash.

For more information about variable substitution, see Section 6.

There are two classes of variables in OJ: local (i.e. local to the current script) and global (i.e., available to all called and calling scripts). For more information, see the discussions of the `local`, `do`, `return`, and `include` commands.

STEP 4: EXECUTION OF THE COMMAND

Generally speaking, a particular portion of the interpreter software is associated with each particular command in the OJ language, and this portion is called only after all three steps outlined above are completed.

This page deliberately left blank.

COMMENTS

Any token beginning with a # is ignored, as are all following tokens on that file line. Therefore, comments are preceded by a #.

IDENTIFIER

All OJ scripts must begin with a comment line; the first character of any OJ script must be #. This is to protect against having a disk savaged by OJ inadvertently executing something other than an OJ script.

CONTINUED LINES

A command line may be continued on the next file line, if it gets too long, by ending the line to be continued with a token consisting of a single backslash. A single command line can thus extend over any number of file lines. For example:

```
# WILL WORK:
set somevar = "This will work just fine " \
   "because I am breaking the line " \
   "on token boundaries."
# WON'T WORK:
set somevar = "This will not work because \
   I am attempting to break up a single token \
   into several lines."
```

WHITESPACE

Tab characters and spaces that do not occur within tokens are regarded as whitespace and are discarded. It is perfectly ok to use any amount of whitespace at the beginning of any command line, as well as between any two tokens. This makes it easy to show program structure, for example, by indenting the commands within an `if`, `loop`, or `switch` region. It is also a good practice to indent continued lines to emphasize the fact that they are continued, as was done in the example above.

This page deliberately left blank.

Variable Substitution

There are really two distinct types of variable substitution at work in OJ. The main type, "dollar substitution," is used for all commands, period. The other type, "automatic substitution," works only in calculational expressions ("calc expression" or "calc exp"), which are only found in certain parts of `if`, `calc`, `loop`, `return`, `exit`, and `setprior` commands. What makes things confusing, perhaps, is that both dollar and automatic substitution are at work in the commands that can contain calculational expressions.

Dollar substitution is made available to the programmer via the `$` character. As the interpreter executes a script, each line of code undergoes tokenization, macro substitution, and then dollar substitution. In dollar substitution, if any token on the line begins with the `$` character, that entire token is replaced by the entire contents of the variable whose name follows the `$`. At this point, the interpreter does not yet care which command is to be executed (i.e., what the first token on the line is); therefore all commands, including the calculational commands, are subject to dollar substitution.

The second form of variable substitution, automatic substitution, is available without the use of the `$` character. It only occurs in calculational expressions, however, and calculational expressions are found in only the following commands: `if`, `calc`, `loop`, `exit`, `return`, and `setprior`.

Here is a complete list of the places where calculational expressions can appear:

`if` *calc_exp*

`calc` *var* = *calc_exp*

`loop` *label* ; *var* = *calc_exp* ; *calc_exp* ; *var* = *calc_exp*

`exit` *calc_exp*

`return` *calc_exp*

`set_prior` *calc_exp*

Please note that no calculational expression can be understood by the interpreter unless it appears in one of the above places. For more information on calculational expressions, see Section 8.

If the command line turns out to be a calculational command, the software which is peculiar to each such command (and which comes into play only after the dollar substitutions have occurred— see Section 4) begins by concatenating all the tokens that appear in the expression.

In the example:

```
calc z = 12 / 14 - (34 * z)
```

the expression on the right side of the assignment operator token (=) will be concatenated into:

12/14-(34*z)

Then, any strings not recognized as real numbers (such as z, above) will be assumed to be variable names, and the values of those variables will be substituted. Thus, the $ is not needed in front of any variable name that occurs in a calculational expression. The value will be substituted automatically.

If the value of z in the example above is 29, then, the expression will become:

12/14-(34*29)

Finally, the entire expression will be evaluated arithmetically, and the resulting real number will become a string of decimal digits (with, in this case, a decimal point). After the calc is executed, the variable z will contain this string. The only other type of character found in the result of an evaluation of a calculational expression is a leading minus sign (-) in the case of a negative number.

SUBSCRIPTS AND ARRAY INDEXES

Subscripts and array indexes can be arbitrary strings; they need not be integers as in many other computer languages. For instance, it is possible to have an ordinary array:

```
z[0]
z[1]
z[2]
```

and it is equally possible to have an array which allows a series of arbitrary strings to be associated with a particular variable name:

```
z[george]
z[natalie]
z[harriet]
```

The elements of these arrays can only be accessed by using their exact names, but one may even have weird multidimensional arrays containing items named, for example:

```
z[george[nose]][wart[0]][itch]
```

The above example is a light-hearted one, but there is plenty of serious applicability for such variables. You can use this feature for data structures such as this:

```
loop 0end ; j = 0 ; j < total_students ; j = j + 1
  fprintf $stdout "student %3d" $j
  fprintf $stdout "name: %s\n" student[$j][name]
  fprintf $stdout "\tgpa: %s\n" student[$j][gpa]
  fprintf $stdout "\tsex: %s\n" student[$j][sex]
  fprintf $stdout "\tclass: %s\n\n" student[$j][class]
0end
```

It is possible to use a variable containing an arbitrary string as an array index, like this:

```
set indexvar = foo          # the content of indexvar is
                            # now the string "foo".
set x[$indexvar] = dahlia
```

We have now created a variable named `x[foo]` and set it to the string "`dahlia`".

USE OF THE DOLLAR SIGN IN ARRAY INDEX SUBSTITUTIONS

Note that the `$` is required in any substitutions made inside square brackets (i.e., when you use a variable as an array index). In array index substitutions, the `$` is required regardless of the particular command you are using.

Below are illustrations of several methods that will work, and several that won't work. All are assumed to be preceded by the following lines of code:

```
set z = 0           # To be used later as an array index.
set y[0] = 123      # Just to create y[0] and put some
                    #   value in it.
          # (Note: the above two lines could have used
          # the calc command instead of the set
          # command with no other changes.)
```

Now, in all the command lines below, we're trying to set the variable x to the value contained in the variable y[0].

These methods will work:

```
calc x = y[0]       # This will work; automatic substi-
                    #   tution is done by the calc com-
                    #   mand.
calc x = $y[0]      # This will work just as well; dol-
                    #   lar substitution occurs before
                    #   the calc command ever sees it.
calc x = $y[$z]     # This will work; dollar substitution
                    #   is done before the calc command
                    #   ever sees it.
calc x = y[$z]      # This will work; automatic sub-
                    #   stitution is done by the calc
                    #   command.
set x = $y[$z]      # This will work. Dollar substitu-
                    #   tion is required because auto-
                    #   matic substitution is not avail-
                    #   able in a set.
```

These methods will not work:

```
set x = y[z]        # This will set the variable "x" to
                    #   the string "y[z]"- not what
                    #   we're trying to do.
set x = y[$z]       # This will set the variable "x" to
                    #   the string "y[$z]"- not what
                    #   we're trying to do.
calc x = y[z]       # This won't work- z won't be sub-
                    #   stituted, and there is no vari-
                    #   able "y[z]". z won't be substi-
                    #   tuted because the $ is always
                    #   required within square brackets.
                    #   OJ will abort.
set x = $y[z]       # This won't work; there is no vari-
                    #   able called "y[z]" and OJ will
                    #   abort.
```

Since an array index can be almost any string, it is fruitless to use a mathematical expression as an array index in the hope that it will be evaluated and the result will become the index value.

For example:

```
set a = z[$x + 1]
```

will not work. However, the following will work and will have the desired effect:

```
calc y = $x + 1       # Expression evaluated here.
                      #   (The $ is optional- both
                      #   dollar and automatic sub-
                      #   stitution are available
                      #   here.)
set a = $z[$y]        # String resulting from expres-
                      #   sion in the calc above is
                      #   now used as an array index.
```

In order to test your understanding of what's going on here, it's a good idea to write a series of scripts using all of the above methods and showing the results. To make things easy on yourself, just put a `show_vars` command at the end of each script.

Here is an example of a script that illustrates a few other points:

```
1  show_tokens_on     # This will cause the interpreter
2                     #   preter to output a veritable
3                     #   torrent of stuff before
4                     #   executing each line.
5  echo_on            # This will cause each line to be
6                     #   echoed to the screen after
7                     #   dollar substitution.
8  set z = 0   # calc z = 0 would have the same effect
9              #   since 0 is a (very simple)
10             #   mathematical expression.
11  set x[0] = george
12  set x[z] = harry
13  fprintf $stdout "\$x[z] == \"%s\"\n" $x[z]
14  fprintf $stdout "\$x[\$z] == \"%s\"\n" $x[$z]
```

NOTE: The backslash appears before two of the dollar signs (in lines 13 and 14) to prevent the interpreter from attempting to perform variable substitution on the tokens of which they would otherwise become the the first character.

The output of the above script is:

```
token[0]: "echo_on"
token[0]: "set"
token[1]: "z"
token[2]: "="
token[3]: "0"
A  8: set z = 0
token[0]: "set"
token[1]: "x[0]"
token[2]: "="
token[3]: "george"
A 11: set x[0] = george
token[0]: "set"
token[1]: "x[z]"
token[2]: "="
token[3]: "harry"
A 12: set x[z] = harry
token[0]: "fprintf"
token[1]: "\$stdout"
token[2]: "\\\$x[z] == \"%s\"\\n"
token[3]: "\$x[z]"
A 13: fprintf 147672 \$x[z] == \"%s\"\r harry
$x[z] == "harry"
token[0]: "fprintf"
token[1]: "\$stdout"
token[2]: "\\\$x[\\\$z] == \"%s\"\\n"
token[3]: "\$x[\$z]"
A 14: fprintf 147672 \$x[\$z] == \"%s\"\r george
$x[$z] == "george"
```

These lines result from show_tokens_on.

. . . This line is from echo_on.

. . . This output is from the first fprintf.

. . . This output is from the second fprintf.

NOTE: Quotation marks surround the control strings in the fprintf commands in order to allow those strings to contain spaces and yet become one and only one token. Please note that the quotation marks surrounding any token are stripped off when the lines are tokenized, as is evident if you look above at the output generated because of the show_tokens_on command. When you want quotation marks to form a portion of the control string, precede them with backslashes.

NOTE: When you look at the output generated by the show_tokens_on command, please note that each token is displayed on your screen as if it had been processed by the alphameric command. The single leading backslash has been converted into two backslashes; otherwise the interpreter might think a single character was meant by the combination of the backslash and the character following it, like \n. For more information about backslash interpretation, see Section 10 and the alphameric command in Section 11.

Calculational Expressions

The expressions found in `calc`, `if`, `loop`, `exit`, `return`, and `set_prior` commands are treated somewhat differently than the arguments of other commands. Variable names do not have to stand alone as individual tokens, and they do not have to be preceded by a `$`; the name will always be substituted for its value (a string) whether it is preceded by a `$` or not (except, of course, for the variable on the left side of the assignment operator token =). For a discussion of this automatic form of variable substitution, see Section 7. In OJ, calculational expressions may appear only in `calc`, `if`, `loop`, `exit`, `return`, and `set_prior` commands. Expressions may be of arbitrary complexity, although the number of levels of parenthesization is currently limited to a depth of 10.

Operands (after variable substitution) must consist entirely of real decimal numbers. The evaluation of a calculational expression is always a real decimal number. Logical operations result in 0 (false) or 1 (true). In the absence of parentheses to the contrary, operators are applied in the order:

!	reverse logical value of next operand or parenthesized expression
*	multiply
/	divide
%	mod (modulus; gives remainder after dividing the integer portion of the operand on the left side of the % by the integer portion of the operand on the right side of the %)
+	add
−	subtract

The following logical operators always result in 1 (true) or 0 (false). Their operands must be real decimal numbers (use the `strcmp()` function to render the comparison of two non-numeric strings into a numeric value):

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to
!=	not equal to

The following logical operators combine multiple logical expressions:

&&	(logical AND)
||	(logical OR)

SPECIAL CALCULATIONAL FUNCTIONS

Calculational expressions may contain the following special functions, whose arguments (in parentheses) may be strings or variable names. Enclose a string with single quotation marks to guarantee that it will be considered a string rather than a variable name. In order to specify a string which contains a single quotation mark character, precede the single quotation mark character with a backslash.

`charval(`*a*`)` returns the integer ASCII value of the first character in *a*. If *a* is a variable which is empty, `charval()` returns 0.

`devrdy(`*device name*`)` returns 1 (i.e., true) if one or more characters are waiting to be picked up from the specified SCF device. For example, `devrdy ('/term')` can be used to test whether any keys are waiting to be picked up from the console. See also the `stdin` and `spilldev` commands.

```
local ctr                   # example
fprintf $stdout "Press any key to continue. \n"
spilldev /term              # throw away anything
                            #   typed in lately
loop 0 ; ; !devrdy('/term') ;
                            # wait for someone to type
                            #   something
    fprintf $stdout "%c" 7          # ring the bell
    loop 1 ; ctr = 0 ; ctr  15 ; ctr = ctr + 1
                            # 15 seconds per ring
        wait 2              # longest wait to break
                            #   is 2 half-seconds
        if (devrdy('/term'))
            break
        endif
     1
0
spilldev /term
```

`frac(`*val*`)` returns the fractional part of *val*. If *val* is 24.31, `frac(`*val*`)` will be 0.31.

`gvarnum(`*varname*`)` returns the element number of the GV_NAME[] (global variable name) and GV_CONT[] (global variable content) arrays which corresponds to *varname*, if it exists, or the element number where *varname* would be if it were inserted alphabetically. You cannot tell whether *varname* exists by using `gvarnum()`; use `length()` for that purpose. See also `lvarnum()`.

```
set flower_dahlias = 29          # example
set flower_daffodils = 34
set flow_control = $in_progress
# ...other variables are set here...      # now
let's find out the number of the
     #   first of the variables whose names
     #   begin with "flower":
set varnum = "" #  create these vars now so
                #  creating them later won't
                #  change the gvarnum() value
set work = ""   #  of any other vars
calc varnum = gvarnum('flower')
substring work = $GV_NAME[$varnum] 0 6
           # (there are 6 chars in "flower")
if !strcmp(work 'flower')
           # if we're here, varnum is the first
           # GV_NAME that begins with "flower",
           # and there is at least one such
           # variable.
 endif
```

`hex2dec(`*val*`)` given a hexadecimal number, returns its decimal equivalent.

`index(`*pattern s [start]*`)` returns the number of the byte where the pattern string occurs within the searched string *s*; or returns -1 if *pattern* is not found in *s*. The value returned counts from the beginning of the string (where 0 is the first byte), regardless of the (optional) start position *start*.

`int(`*val*`)` returns the integer part of *val*. If *val* is 24.31, `int(`*val*`)` will be 24.

`isdir(`*path*`)` returns 1 if *path* is an extant directory, 0 if it does not exist or it is not a directory.

`isfile(`*path*`)` returns 1 if *path* is an extant file, but not a directory. Otherwise returns 0.

`isreal(`*string*`)` returns 1 if *string* is some real number (e.g.: 1, -12.3, .3314, etc.), or 0 if it is not (e.g.: 23-44, 1.2.a, Fred).

`length(`*var*`)` returns the length of the string contained in *var*, or -1 if there is no variable named "*var*." This function can be used to determine whether a variable exists.

`lvarnum(`*varname*`)` returns the element number of the LV_NAME[] (local variable name) and LV_CONT[] (local variable content) arrays which corresponds to *varname*, if it exists, or the element number where it would be if it were inserted alphabetically. You cannot tell whether *varname* exists by using `lvarnum();` use `length()` for that purpose. See also `gvarnum()` for more information and an example.

`max(`*varname varname*`)`
`min(`*varname varname*`)` yields the larger (smaller) of the two values.

`namecmp(`*target pattern*`)` compares two filename strings to see if they qualify as a match in the OS-9 world. The *target* string is an actual filename. Two metacharacters are recognized in the *pattern* string: ? matches any single character and * matches any string of characters. Upper/lower case distinctions are always ignored Returns 0 if there is a match, or −1 if not.

`oct2dec(`*val*`)` given an octal number, returns its decimal equivalent.

`rand()` returns a 31-bit pseudo-random positive integer. (See also the `setrand` and `randlist` commands.) To obtain a random integer with 11 different possible values in the range 0 to 10, inclusive, use:

```
rand() % 11
```

`strcmp(`*a b*`)` returns a negative number if string *a* is lexicographically less than (i.e., prior to) string *b*; a positive number if *b* < *a*; or 0 if the strings are identical.

System variables are used just like all other variables (see Sections 7 and 8).

ERROR REPORTING

`e_banner`

is a pseudo-variable containing the time and date of this particular invocation of OJ, the invocation name of OJ itself, the invocation name of the current script, the current line number in that script, and the `do` stack of calling scripts, if any. For example:

```
90/05/19 15:51:06 oj {intcept_handler()}:
  OJ interrupted by signal 3. line 9 of
    script
"/dd/USR/SCRIPTS/massedit.oj"
```

It is often good to use this in an `fprintf` (possibly to `$stderr`) just before `exiting` on account of some error condition.

`qreturn`

is a local variable which contains an integer value returned by a number of OJ commands. Its value should always be checked immediately after any command which sets it, lest some succeeding command set it to another value before it is checked. You can use either `switch` or `if` to check `qreturn`, as neither `switch` nor `if` sets `qreturn`. The values of `qreturn` are interpretable by system pseudo-variable `e_msg[$qreturn]`. They are also listed in Section 13. Generally speaking, if all went well, `qreturn` >= 0; if not, `qreturn` is some meaningful negative value.

`E_NONINT`, etc. (see Section 13)

are error condition constants available as global variables, containing integer values. These should be treated as constants to compare with the value of `qreturn` to check for specific error conditions reported by `qreturn`. For a complete list of these, see Section 13. Instead of doing this:

```
if (qreturn == -2)
```

do this:

```
if (qreturn == E_NONINT)
```

because the exact value of E_NONINT may change in future versions of OJ, whereas the meaning of E_NONINT will not change.

`e_msg[$qreturn]` (see Section 13) is a pseudo-array of variables containing brief messages, one per array element, explaining the meaning of the value of `qreturn` used as the index value. Each message also contains the CONSTANT name of the particular value of `qreturn`, e.g.:

```
E_NONINT: an integer value was
        required; this wasn't one
```

INPUT/OUTPUT

`stdout,stderr` are output file handles for `writefile` and `fprintf`. Unless OJ's own standard output and/or standard error stream has been redirected elsewhere, both will go to the console screen. Both are global variables.

MEMORY MANAGEMENT AND DATABASE-LIKE OPERATIONS

(see also the system functions `gvarnum()` and `lvarnum()` in Section 8)

LVARS is the number of local variables currently in existence.

GVARS is the number of global variables currently in existence.

LV_NAME[*n*] is a way of accessing the names of all the local variables. The index must be an integer in the range 0 to (LVARS - 1).

GV_NAME[*n*] is a way of accessing the names of all the global variables. The index must be an integer in the range 0 to (GVARS - 1).

LV_CONT[*n*] is a way of accessing the contents of all the local variables. The index must be an integer in the range 0 to (LVARS - 1).

GV_CONT[*n*] is a way of accessing the contents of all the global variables. The index must be an integer in the range 0 to (GVARS - 1).

MISCELLANEOUS

`argc` — is a local variable containing the number of arguments (`argc` stands for "argument count") passed to the script, either at OJ invocation time or via a `do` command. The script name itself counts as one argument, so the value of `argc` is always at least 1.

`argv[0...n]` — is the arguments passed to the current script. `argv[0]` is the name of the script (but only if it is the original or the `including` script— the name of any `included` script exists only in the variable `curscript` [see below]). `argv`'s index value has a range of 0 to (`argc`–1). (`argv` stands for "argument vector"; in computerese a "vector" is usually a list of some kind). The array elements of `argv[]` are all local variables.

`curscript` — gives the name of the current script, whether it is an `including` or an `included` script.

`exitstat` — is the exit status value of the most recent OS-9 command, or the value `returned` by a called (i.e. `done`) script to the calling (i.e. `doing`) script. `exitstat` is useful when you've used a hyphen before an OS-9 system call (an invocation of any regular OS-9 program) to prevent OJ from aborting on a non-zero exit status from that invocation. (See the *any_OS-9_program_invocation_name* command at the beginning of Section 11.) For a discussion of the exit status of a `done` script, see the descriptions of `do` and `return`. `exitstat` is a local variable.

`systime` — is a pseudo-variable containing the date and time current to the second, in the form *yy/mm/dd hh:mm:ss*. The hour number is given in military (i.e. 24-hour) form. For example, March 8, 1989 at 12:38:02 A.M. would appear as `89/03/08 00:38:02`. The same day at 8:32:00 P.M. would appear as `89/03/08 20:32:00`. This is not really a variable at all, but you use it just as if it were one.

`systimef` — is the same as `systime`, but without punctuation, for use in filenames. E.g.: `890308203200`

`procid` — is a global variable containing the OJ interpreter's process ID in exactly five digits (with leading zeros, if needed).

`pwd` is a pseudo-variable containing the current working data directory, expressed as a full pathname beginning with an explicit device name.

`userid` is a global variable containing the current user's group/owner id number, in the form `255.254` if group is 255 and owner is 254.

SOCKET INTERACTION

`reply[]` (See `send` command description.)

`ereply[]` (See `send` command description.)

`sendbuf[]` (See `send` command description.)

Backslash-Interpreting Commands

In "backslash-interpreting commands," backslashes indicate special (often non-printable) characters in strings; this usage is generally similar to that used in the C language.

In the backslash-interpreting commands, token strings may contain any of the following special conversions to single byte values:

String	Byte value	meaning
\r	0dH	carriage return
\n	0dH	another way to say carriage return
\l	0aH	linefeed
\f	0cH	formfeed
\t	09H	tab
\b	08H	backspace
\\	5CH ('\\')	a backslash
\"	22H ('\"')	a double quotation mark (double quotation marks are otherwise assumed to surround tokens and will be discarded during tokenization)
\$	24H ('$')	allows a token to begin with $ without forcing variable substitution
\#	23H ('#')	allows a token to begin with # without initiating a comment
\d001	01H	decimal value of 1
\001	01H	octal value of 1
\x01	01H	hexadecimal value of 1

NOTE: as character substitutions in hard strings are done before variable substitutions, no variable substitution will occur if the original token began with \$. This means that \$*var* will become $*var* but but will not undergo variable substitution.

NOTE: Use 3 digits for decimal and octal and 2 digits for hex, even if you don't need them. This prevents subsequent valid characters from being treated as part of the value of the byte.

The backslash-interpreting commands include `arraytok`, `fprintf`, `send`, `set` (note especially the =R option), `set_mbf`, `sprintf`, `replace`, and `writefile`. The `alphameric` command performs backslash-deinterpretation.

external process control — ***any_OS-9_program_invocation_name***

format: (Just like any OS-9 shell command line, with a few differences. See the documentation on the Microware "shell" program in the OS-9 user manual, and the notes below.)

purpose: The tokens, after variable substitution, if any, will be concatenated into a string which will become a command line passed to the OS-9 "shell" program. Execution of the script will halt until the program has finished running. Example:

```
dir /h0 > $fn
```

will cause `dir` to be run on `/h0`, and the standard output to be written on the file whose name is stored in variable `fn`.

notes: (1) Concatenation of tokens is not handled the same way as in all other OJ commands. The separate tokens are concatenated into a string with spaces between them, except that redirection operators are snugged up against the tokens they modify.

(2) Redirection operators (< , > , !, >>) must be single tokens; they should be followed by whitespace, even though this is not normal OS-9 practice. This is to allow variable substitution to take place in the normal way.

(3) If the exit status of the program is non-zero, OJ will abort immediately, unless the command is immediately preceded by a dash (-). For example:

`-del e` will not abort the script even if `e` does not exist and therefore `del` returns non-zero status. Afterwards, the exit status may be found in the system variable `exitstat`.

(4) Invocations of OJ itself are automatically altered in such a way as to cause certain OJ modes to propagate to the forked OJ process (see `oj` command).

(5) This command sets `qreturn`; check for E_PSNOFORK, E_PSWENTAWAY values (see Section 13).

(6) See also the `redirect` command. If a `redirect` is in effect, the OS-9

"shell" program does not mediate the invocation, and redirection operators may not be used.

(7) For interactive control of a forked process, use `socket` instead. See `socket`, `send`, etc.

(8) There must be a one-to-one correspondence between the arguments to the invoked program and the tokens used to create those arguments. For example, the following example will not work, because instead of getting two arguments, "`-d`" and "`-l`", `ls` will get only one strange argument, "`-d -l`":

```
set arg1 = "-d -l"
ls $arg1
```

On the other hand, this will work fine:

```
set arg1 = "-d"
set arg2 = "-l"
ls $arg1 $arg2
```

It is sometimes extremely inconvenient to make a one-to-one correspondence between tokens and arguments. For that reason, a special format is provided for strings which contain multiple arguments, separated by whitespace, that you want to have parsed into separate arguments. The trick is this: begin such a single string with a # character. (This requires the use of a backslash when first setting a variable to it, so that it is not seen as a comment-begin delimiter.) For example, this will work:

```
set arg1 = "\# -d -l"
ls $arg1
```

The fact that the above method is supported by OJ makes it impossible to create a single argument string for an OS-9 invocation which both begins with a # and also contains spaces. It seems unlikely that anyone will need to do this very often, though. Note also that the pound sign in OS-9 is used to specify the stack space of the forked process. Since it will be stripped, use two of them:

```
foo \#\#58      # 58 k of stack space.
```

string manipulation

alphameric

format: `alphameric` *var_to_set* = *string_1* (...*string_n*)

purpose: Mainly for making debug displays; replaces non-alphameric bytes, such as 0dH, with backslash-preceded equivalents, such as `\r`. Uncommon byte values are given in hex, as, for example, `\x00` for a null byte.

notes: `alphameric` reverses what happens to hard strings in backslash-interpreting commands (see Section 10). See also the `set` command in Section 11, which has approximately the opposite effect of `alphameric` (especially the =R option of `set`).

arraytok

string manipulation

format: `arraytok` *separator_character(s) arrayname (option) string_1 (...string_n)*

(NOTE: as with all commands, this one must be written all on one line in the script. If it won't fit, use the backslash line-continuation feature [see Section 6].)

option: `-GOBBLE` or `-NOGOBBLE` (default)

purpose: Create an array of variables each containing a token from the line.

notes: First, *string_1 . . . string_n* are concatenated. Then the single long string thus concatenated is re-divided into tokens wherever any separator character appears. All separator characters are discarded. Each token is then stored in an element of an array of variables whose name is *arrayname*`[0]` . . . *arrayname*`[n]`. The variable *arrayname*, without an index, contains the number of valid elements in the array. To cause the array to be created as local variables, precede `arraytok` with a

```
local arrayname
```

If no option is given, or if the default option `-NOGOBBLE` is (redundantly) specified, each appearance of the separator character will cause a token to be created, even if no other (i.e., non-separator) characters appear between them. This is useful in cases where a line of data is separated into fields by field separator characters, and it is necessary to render even empty fields into numbered tokens.

Alternatively, if the `-GOBBLE` option is specified, wherever one separator character immediately follows another (with nothing to tokenize in between) no empty token will be created. This is useful if, for example, it is necessary to divide a line of text into separate tokens, and discard all spaces and tabs between them.

Here are three examples:

Example 1:

```
set path = /h0/USR/SRN/TOOLBOX/OJ
arraytok / pathitem $path
fprintf $stdout "pathitem == %d\n" $pathitem
loop 1 ; ctr = 0 ; ctr < pathitem ; \
    ctr = ctr + 1          # rest of loop command
    fprintf $stdout \
        "pathitem[%d] == %s\n" $ctr
$pathitem[$ctr]
1
```

Output from code in example 1:

```
pathitem == 6
pathitem[0] == ""
pathitem[1] == "h0"
pathitem[2] == "USR"
pathitem[3] == "SRN"
pathitem[4] == "TOOLBOX"
pathitem[5] == "OJ"
```

Example 2:

```
set a = "the quick"
set b = "fox"
arraytok " " foo $a brown " " $b \
   "jumps  over the lazy dog."
     # there are 3 spaces between the
     # words "jumps" and "over"
fprintf $stdout "foo == %s\n" $foo
loop 0 ; ctr = 0 ; ctr < foo ; ctr = ctr + 1
   fprintf $stdout "foo[%d] == \"%s\"\n" $ctr \
     $foo[$ctr]
0
```

Output from code in example 2:

```
foo == 9
foo[0] == "the"
foo[1] == "quickbrown"
foo[2] == "foxjumps"
foo[3] == ""
foo[4] == ""
foo[5] == "over"
foo[6] == "the"
foo[7] == "lazy"
foo[8] == "dog."
```

Example 3:

```
# just like Example 2 except -GOBBLE option is
# added to arraytok, 2 tab characters are added
# after "jumps", and spaces and tab characters
# are identified as separator characters:
arraytok " \t" foo -GOBBLE $a brown " " $b \
    "jumps\t\t   over the lazy dog."
```

Output from code in example 3:

```
foo == 7
foo[0] == "the"
foo[1] == "quickbrown"
foo[2] == "foxjumps"
foo[3] == "over"
foo[4] == "the"
foo[5] == "lazy"
foo[6] == "dog."
```

Note that the empty fields were "gobbled up"— i.e. they disappeared.

As is obvious from the above Example 3, `arraytok` is a "backslash-interpreting" command.

string manipulation

bits

format: `bits` *var* = *decimal_integer*

purpose: Store the string of thirty-two 1's and 0's representing the condition of the bits of a 32-bit integer whose value is the decimal value given.

notes: Four tokens.

break

flow of control

format: `break`

purpose: (1) Immediately cease iterating the nearest enclosing `loop`, and restart the program at the ending label of that `loop`.

(2) Proceed immediately to the statement label at the end of the nearest enclosing `switch`, skipping all the rest of the `cases`, etc.

notes: Exactly 1 token. Not a valid command unless there is a `loop` or `switch` in progress to `break`.

calculation **calc**

format: `calc` *resultvarname* = *calc_expression*

purpose: Make a calculation and store the result in *resultvarname.*

notes: See Sections 7 and 8. Variables whose names appear in the expression will undergo substitution even without the use of the $ character. This can be prevented by using single quotation marks.

case

flow of control

format: `case` *string_to_match_in_a_switch*

purpose: See `switch`.

notes: Only valid within a `switch` region.

miscellaneous **cd, chd**

format: `cd` *directory_to_become_current_data_directory*

or

`chd` *directory_to_become_current_data_directory*

purpose: Change current data directory.

notes: See the chd command (actually the chdir() function) in the OS-9 user manual. `qreturn` will be 0 if the command was successful. There is no difference between `cd` and `chd`.

chx

miscellaneous

format: `chx` *directory_to_become_current_execution_directory*

purpose: Change current execution directory.

notes: See the chx command (actually the chxdir() function) in the OS-9 user manual. `qreturn` will be 0 if the command was successful.

input/output

closefile

format: `closefile` *file_handle*

purpose: Close a file which is already open, thus writing to disk any buffered data which hasn't been written there yet, and releasing the OS-9 path table slot associated with it.

notes: Don't forget to use a $ before the file handle variable. Sets `qreturn` to E_NO_ERR (0) if all went well.

See also `openfile`, `readfile`, `writefile`, `fprintf`, `seekfile`, and `tellfile`.

continue

flow of control

format: `continue`

purpose: Immediately start the next iteration of the nearest enclosing `loop`, without executing the remainder of the current iteration.

notes: Exactly 1 token. Not a valid command unless there is a `loop` in progress to `continue`. See `loop`.

debugging

debug_on (debug_off)

format: `debug_on`

or

`debug_off`

purpose: Turn on (off) the debugger.

notes: Inserting a `debug_on` in a script has the same effect as setting a debugger breakpoint on the following executable line. See Section 12.

default

flow of control

format: `default`

purpose: See `switch`.

notes: Only valid within a `switch` region.

time measurement **difftime**

format: `difftime` *target_varname* = *marktime_value_1* *marktime_value_2*

(NOTE: as with all commands, this one must be written all on one line in the script. If it won't fit, use the backslash line-continuation feature [see Section 6].)

purpose: Write on the *target_varname* the number of seconds, including any fraction of a second, between the time when *marktime_value_1* was set by `marktime` and the time when *marktime_value_2* was set by `marktime`.

notes: In general, if predictability and/or accuracy are critical, try to avoid disk input/output between one `marktime` and another. With care, accuracy levels in the neighborhood of 0.1 second should be attainable. There are several possible uses for this command, including experiments intended to show which of several OJ programming algorithms is fastest. When performing such experiments, it is important that the system is otherwise idle, and that the code be iterated many times. Note that `marktime` and `difftime` can not be used to measure processing time, only elapsed real time.

For example:

```
#
marktime time0      # start timing now
   # something happens here and you want to
   # know how long it takes.
marktime time1  # make a note of the time here.
difftime elapsed = $time0 $time1
fprintf $stdout "It took %.2f seconds to do it.\n"
\
   $elapsed
```

See also `marktime` and `julian`.

do

flow of control

format: `do` *OJ_script_name (arg_1...arg_n)*

purpose: Execute the named script as a subroutine.

notes: The arguments passed to the subroutine, if any, will appear as `argv[1...n]`; the subroutine's name will appear as `argv[0]`.

The subroutine script will share all the global variables already in existence with the calling script, and any global variables it creates will still be around when the subroutine returns. It will, however, have its own independent bank of local variables, including the system variables `argc`, `argv[0...n]`, `exitstat` , and `qreturn`. See the `local` command.

The called script and the calling script will not know about each other's statement labels or `loops`; i.e., you may not have a `goto`, `switch`, or `loop` in one script which refers to a label in another script.

The called script's `return` value is available to the calling script as `exitstat` (see the `return` command).

The degree to which you can nest `do` commands is limited by the size of the OS-9 file stream table, which usually can have 32 entries. All open files use up one entry each. This means that if you write a script that calls itself recursively via `do`, it can recur at most 31 times before a "path table full" error aborts OJ altogether.

debugging

echo_on (echo_off)

format: `echo_on`

or

`echo_off`

purpose: The single most useful script debugging command. Causes OJ to output each command line to be executed (does not show unexecuted command lines) after variable substitution, but before execution. (Echo output is made to the console via stderr.)

notes: Exactly 1 token. The first item on each output line is a letter; A means the invocation script, B the script called by the A script, C the script called by the B script, etc. After this letter comes the line number, followed by the line itself.

The functionality of `echo_on` and `echo_off` are available in the interactive debugger via the `e1` and `e0` commands (see Section 12).

else

flow of control

format: `else`

purpose: (See `if` command)

notes: Exactly 1 token. If you don't need an `else`, you don't have to use one. Requires prior use of an `if`.

flow of control

endif

format: `endif`

purpose: (See `if` command.)

notes: Exactly 1 token. Requires prior use of an `if`.

exit

flow of control

format: `exit (calc_expression)`

purpose: Terminate OJ altogether and, optionally, set the exit value of the OJ interpreter.

notes: 1 or 2 tokens. If used, the optional second token must evaluate to an integer. If there is no second token, OJ's exit status will be 0. See Section 8.

input/output

fprintf

format: `fprintf` *file_handle control_string (string_1...string_n)*

purpose: Write a formatted string on a file (or `$stdout` or `$stderr`) using the same control string and conversion conventions as the fprintf() function in the C language.

notes: Please refer to Kernighan and Ritchie's The C Programming Language or any other good book on C. To output text to the screen, use

```
fprintf  $stdout ...
```

or

```
fprintf $stderr ...
```

`fprintf` is a "backslash-interpreting" command.

See also `openfile`, `closefile`, `readfile`, `writefile`, `seekfile`, and `tellfile`. See also `sprintf`.

goto

flow of control

format: `goto` *label*

purpose: Resume execution elsewhere in the script.

notes: If at all possible, this command should be avoided. It has the effect of cancelling all `switches` and `loops` which may be in progress. Its only proper function is to allow there to be only one place within a script to be jumped to prior to an on-error `exit` or `return`, so that error reporting code need not be duplicated all over the place. See the *label* command.

flow of control **if**

format: `if` *calculational_expression*

purpose: Conditionally control script execution.

notes: Used with `else` and `endif` in the usual fashion. If the expression is evaluated as non-zero, it will be regarded as true.

2 or more tokens. Requires the use of `endif` to define its region of effect. The use of `else` is optional.

For example:

```
if (calc expression evaluating non-zero)
   # commands in this region
   # will be executed
else
   # commands in this region
   # will not be executed
endif
```

See also Section 8.

ignore

socket interaction

format: `ignore` *socket_handle*

purpose: Discard all bytes received from the `socketed` device or process during the next `send`.

notes: `ignore` is generally used only when the number of bytes is expected to be very large and occupy too much memory uselessly.

`ignore` affects only the very next `send` to the specified *socket_handle*. One effect of `ignore` is that all of the waits specified by any preceding `set_waits` will be fully satisfied before the `send` is completed.

See also `send`, `set_waits`.

flow of control

include

format: `include` *oj_script_name*

purpose: Exactly the same as `do`, except that an `included` script has the same local variables as the including script. In other words, the scope of the `including` script's local variables is increased to encompass the included script, and so in a way, those local variables are globalized. This is useful for passing `argc` and `argv[0...n]` through to a called script, for example.

notes: Always exactly 2 tokens. No arguments can be passed. The command is called "include" because it is as though the `included` script's code is inserted wholesale into the `including` script's code. However, as in `do`, the `including` script's `exitstat` can be set by using the `return` command in the `included` script.

istop (istart)

flow of control

format: `istop`

or

`istart`

purpose: Turns off (on) the interpreter unconditionally for any number of lines. Allows documentation to appear within a script without the necessity of beginning each line with a comment character (#).

notes: Once an `istop` has been encountered, the only way to restart interpretation is with an `istart`. If the interpreter is already enabled, `istart` has no effect.

time measurement

julian

format: `julian` *var_for_day_number var_for_second_number var_for_tick_number var_for_ticks-per-second_value = marktime_value*

(NOTE: as with all commands, this one must be written all on one line in the script. If it won't fit, use the backslash line-continuation feature [see Section 6].)

purpose: Display exact date and time (to the clock tick) when a `marktime` was executed, in Julian format.

notes: The day number will be a Julian day number. The following information, quoted from Webster's Third New International Dictionary, is relevant:

> "A Julian period is a chronological period of 7980 Julian years that combines the solar and lunar cycles and the Roman indiction cycle and is reckoned from the year 4713 B. C. when the first years of these cycles coincided.
>
> "A Julian year is exactly 365 days, 6 hours, adopted in the Julian calendar.
>
> "A Julian day number is the number of a day in the Julian day calendar (as 2,436,934 for January 1, 1960)."

The second number will be the number of the second since midnight. There are 86,400 seconds in a day.

The tick number will be the number of ticks since the beginning of the second. The ticks-per-second value is just what it says it is. This value is provided because different OS-9 systems may have different numbers of ticks per second. See `marktime` and `difftime`; see the `systime` and `systimef` system variables if Julian dates are not what you want.

For example:

```
marktime timenow
julian day sec tick tps = $timenow
fprintf $stdout \
  "day is %s; sec is %s; tick is %s; %s
ticks/sec\n" \
  $day $sec $tick $tps
```

label

flow of control

format: *label*

purpose: Mark a position in the program at which a `goto` command can restart execution, or at which a `loop` or `switch` region ends.

notes: Labels must begin with a decimal digit (0-9), and thereafter they (like variable names) must consist entirely of alphabet characters (both upper and lower case), numeric characters (0 through 9), periods (.), and underscores (_). No two labels can be the same in any one script. A valid label can have only one token on the line. A label can be any length.

memory management

local

format: `local` *varname_1* (...*varname_n*)

purpose: Force all the *varnames* to be local variables henceforth within this script (but not within any `done` or `doing` scripts).

notes: There are two classes of variables in OJ. One is called "global," and the other is called "local." Both kinds work the same way, and they are used in the same ways. Indeed, you can ignore the distinction entirely, in which case all your variables will be global, except for the system variables `qreturn`, `exitstat`, `argc`, and `argv[0...n]`, which are always local regardless of anything you may do.

Global variables are created automatically whenever you set them for the first time. They are available thereafter throughout the entire invocation of OJ, in all called and calling scripts. Similarly, any global variables created and/or set in any scripts called via the `do` command will continue to exist and retain their new values even after the `done` script has returned to the `doing` script.

Local variables are created by the `local` command. Any script which uses local variables should declare them at the beginning of the script with one or more `local` commands; this practice will make the script easier to maintain. However, it is ok to use `local` before any other mention of the variable, even if the `local` does not appear at the beginning of the script.

Whenever a variable is mentioned by name, OJ looks to see whether it exists. It always looks first in the bank of local variables, and if it is not there, it looks in the bank of global variables. If the variable does not exist and must be set to some value, the variable is created as a global variable.

The only way to create a local variable is to use `local`. The effect of a `local` is to create the named variable(s) in the bank of local variables, with no content (i.e., with null strings) in them. Thereafter, OJ will find them there first and not even bother to look in the bank of global variables for them.

When a called script terminates (encounters a `return` or comes to an end), its local variables go away, never to be seen again. It is extremely useful to have variables that are "local" to a script so that it can call itself recursively without the various recursions overwriting each other's data, and so that the memory allocated to its local variables can be recovered automatically at its end. When a script with local variables calls another script with local variables, even

if those local variables have the same names, there will be no interference between the two scripts. In general, it is advisable to use local variables, and write your scripts as modules that can call one another via do. Since the local variable bank is always consulted first, your scripts will run faster.

Here is a demonstration of the use of local variables:

```
# demo.oj begins here
fprintf $stdout "Type a string > "
stdin astring          # user types a string here
do capitalize $astring bstring
                       # call capitalize.oj
if exitstat
   exit $exitstat          # an error occurred in
                           # capitalize.oj
endif
fprintf $stdout \
   "The capitalized string is: \"%s\"\n" $bstring
fprintf $stdout \
   "The original string was: \"%s\"\n" $astring
# demo.oj ends here

# capitalize.oj begins here
local astring     # this will not be confused with
                  # the global astring, if there
                  # is one (which there is).
if (argc != 3)
  fprintf $stderr \
    "\"%s\" requires 3 args; you provided  %d." \
        $argc
  return 1            # a general on-error value
endif
set astring = "X"  # "astring" in "demo.oj" will
             #  not be overwritten by doing this.
set $argv[2] =U $argv[1] # capitalize first arg
                         # and set the global var
                         # in the second arg.
return 0                # this is not really needed
                        # since this is the end.
# capitalize.oj ends here.
```

See also vsave, vrestore.

flow of control

loop

format: `loop` *end_label* ; (*init_var* = *initialization_calc_expression*) ; (*test_calc_expression*) ; (*increm_var* = *incrementing_calc_expression*)

(NOTE: as with all commands, this one must be written all on one line in the script. If it won't fit, use the backslash line-continuation feature [see Section 6].)

purpose: Keep looping (i.e., repeatedly execute the code between the `loop` command line and its ending label) until the *test_expression* evaluates to 0, or a `break` command is encountered.

notes: Everything is optional except the three semicolons and the *end_label*. The semicolons must always be present, and they must be surrounded by whitespace (i.e., be separate tokens). See also `break`, `continue`, and *label*. A `loop` is executed in the same way that "for" loops are executed in the C language: first the *initialization_calc_expression* is evaluated and the result is stored in *init_var*. Then the *test_calc_expression* is evaluated. If the *test_calc_expression* evaluates as 0, the program recommences at the loop's *end_label*, and the `loop`'s contents, i.e., the OJ command lines which lie between the `loop` command and the line on which its *end_label* appears, are not executed. If the *test_calc_expression* evaluates non-zero, the contents of the loop are executed. When the *end_label* is encountered, the *incrementing_calc_expression* is evaluated and the result is stored in *increm_var*. Then the *test_calc_expression* is re-evaluated. If the *test_calc_expression* evaluates 0, the `loop` is terminated. In other words, the code within the `loop` will be executed iteratively only as long as the *test_calc_expression* evaluates non-zero. See Section 8.

Usually, of course, *init_var* and *increm_var* will be one and the same variable. Following is an example of a typical `loop` which will be iterated ten times; on the first time through the `loop`, the variable `loopvar` will have a value of 0, and on the last iteration `loopvar` will have a value of 9.

```
# a plain vanilla loop
loop 0432this_is_the_end_label ; loopvar = 0 ; \
    loopvar < 10 ; loopvar = loopvar + 1
    fprintf $stdout "loopvar == %d.\n" $loopvar
0432this_is_the_end_label
```

marktime

time measurement

format: `marktime` *varname*

purpose: Record the current date and time, accurate to one OS-9 clock tick (often 1/128 of a second).

notes: This command is used with `difftime` to measure an interval of elapsed time. This is useful for optimization of procedures, for checking system loading, etc.

See `difftime` and `julian`. The string stored in the variable is interpretable only by `difftime` and `julian`. There is no point in displaying it; it is not a character string.

flow of control **oj**

format: `oj` *oj_script_name (arg_1...arg_n)*

purpose: Re-invokes OJ as a subprocess, automatically propagating operating modes as shown in the following table:

if this mode is in effect:	then the argument automatically appended will be:
`--d` (debug mode)	`--d` (unless backgrounded with &)
`--e` or `echo_on`	`--e` (unless backgrounded with &)
`echo_off`	none
`--v` or `verbose_on`	`--v` (unless backgrounded with &)
`verbose_off`	none
`--x` or `sys_exec_off`	`--x`
`sys_exec_on`	none
`--q` (quiet mode)	`--q`

(For information about OJ invocation parameters [such as --q], see Section 4).

notes: This command is generally not what you want unless you want to fork OJ as a background process. See the `do` command, which has far less overhead and which allows the sharing of global variables between the calling and the called script.

This is actually handled just like any other OS-9 process invocation (see *any_OS-9_program_invocation_name* at the beginning of this Section), except for the forced inheritance of the operating parameters described above. The current OJ process's variables and the subroutine's variables are totally separate and independent of each other. This command sets `qreturn`; you might want to check it for equality to the E_PSNOFORK and E_PSWENTAWAY error values.

openfile

input/output

format: `openfile` *var_for_file_handle = file_name action*

purpose: Open a disk file for reading, writing, etc.

notes: If the file could not be opened for some reason, the *var_for_file_handle* will be set to '0' and `qreturn` will be nonzero. If all went well, `qreturn` will be E_NO_ERR (0). All other file i/o commands require the file handle value placed in *var_for_file_handle* by `openfile`. *actions* are as follows:

`r`	open for reading. /Useful for checking to see whether a file exists.
`w`	open for writing. Current contents are lost. Creates a file if necessary.
`a`	append (write) at end of file, creating a file if necessary.
`r+`	read from and write to the file. Nondestructive of existing contents until you overwrite them with a `writefile` command.
`w+`	read from and write to the file. If the file exists, its contents are destroyed; if it does not exist, it is created.
`a+`	read anywhere in the file, but only write at the end of it.

For output to stdout or stderr, no `openfile` is necessary. `$stdout` and `$stderr` are the file handles for the standard output and standard error streams.

See also `readfile`, `writefile`, `closefile`, `fprintf`, `seekfile`, and `tellfile`.

socket interaction

password

format 1: `password`

format 2: `password = string`

purpose: Set the password string. This string is useful only in a `send` (see `send`).

notes: If the first format is used, the user is interactively asked to type in the password. The password will not be displayed or echoed to the screen. The user will be asked to type the password twice at pairs of prompts until the two strings match. If the second format is used, the password is *string*. The token `$PASSWORD` in any subsequent `send` will be replaced by this string. The password string is not a true variable: it cannot be used in any command other than `send`. Passwords are therefore secure except from users who can dump memory, or tap into the stream emerging from some serial device, or who can subvert a `socketed` process in some way.

randlist

random numbers

format: `randlist` *arrayname arraysize lowest_possible_value number_of_possible_values* (*option*)

(NOTE: as with all commands, this one must be written all on one line in the script. If it won't fit, use the backslash line-continuation feature [see Section 6].)

options:

`UNIQUE` No two array elements may contain the same value.

`REPEATABLE` The same value may appear any number of times within an array.

purpose: Create an array of variables whose name is *arrayname*, with a number of elements equal to *arraysize*, and place in each one a pseudo-random integer whose least possible value is *lowest_possible_value* (which may be a negative integer) and whose greatest possible value is ([*lowest_possible_value* + *number_of_possible_values*] - 1).

notes: If a `UNIQUE` is in effect, and *number_of_possible_values* is less than *arraysize*, OJ will abort. See also the `rand()` function in Section 8 and the `setrand` command. If *arrayname* is a local variable, `randlist` will create the entire array as local variables. Otherwise, globals will be created.

format: `readfile` *file_handle* *var_for_line_from_file*

purpose: Place a newline-terminated line from the file into a variable. The newline will not be present at the end.

notes: `qreturn` will be less than 0 if an error occurred or if there are no more data in the file (in which case the `qreturn` value will be E_RFEOF). Otherwise, `qreturn` will be a positive value equal to the number of bytes read.

Don't forget the $ before the file handle variable. Don't attempt to read files which are not organized as lines of text. The `readfile` will continue until a newline byte is encountered, which may never happen. In such a case, the variable given as the third token may become quite enormous and use up all available memory, or simply abort OJ altogether with an "insufficient memory" error.

See also `openfile`, `closefile`, `writefile`, `fprintf`, `seekfile`, and `tellfile`.

redirect

external process control

format: `redirect stdin` *varname*

purpose: The standard input stream to the next OS-9 program invoked by the script will be the contents of the variable *varname*. *varname* should exist and should have something in it.

format: `redirect stdout` *varname*

purpose: The standard output stream from the next OS-9 program invoked by the script will be to the variable named *varname*. When such an invocation takes place, the variable will first be initialized to a null string. This means that the contents of the variable will always be only the output of the OS-9 program.

format: `redirect stderr` *varname*

purpose: This is just like `redirect stdout` *varname*, except, obviously, it redirects `stderr` instead of `stdout`.

notes: Don't use a dollar sign before stdout or stderr; here they are keywords, not file handles. Normally, one probably would not want to use a $ with *varname* either, unless the name of the variable to be used is the content of *varname*.

If an OS-9 program invocation is not preceded by a `redirect`, OJ simply passes the whole string which comprises the invocation to the OS-9 "shell" program, which parses the string just as if it were typed at an OS-9 prompt, and then the shell program actually performs the invocation. However, after a `redirect` *somestream somevar*, the subsequent OS-9 program call is not mediated by the shell program. Therefore, if a `redirect` is in effect, redirection operators (< , > , ! , >>) may not be used. See also *any_OS-9_program_invocation_name* at the beginning of Section 11.

string manipulation

replace

format: `replace` *code:_S_or_C char(s)_to_replace what_to_replace_them_with output_var = string_1 (...string_n)*

(NOTE: as with all commands, this one must be written all on one line in the script. If it won't fit, use the backslash line-continuation feature [see Section 6].)

purpose: Replaces characters or strings within strings. First, `replace` concatenates *string_1 . . . string_n* into a single string, and then:

If code is C: All instances of each of the *char(s)_to_replace* in the concatenated string are replaced by the string *what_to_replace_them_with.*

If code is S: All instances of the entire string *char(s)_to_replace* in the concatenated string are replaced by the string *what_to_replace_them_with.*

Then the resulting string becomes the content of *output_var.*

notes: Must be at least 7 tokens. This is a "backslash-interpreting" command.

Examples:

```
replace S yellow green \
   var = "I like the yellow ones."
            # $var now contains string:
            # "I like the green ones."
replace C " e" "X" \
    var = "I like the yellow ones."
            # $var now contains string:
            # "IXlikXXthXX yellowXonXs."
```

return

flow of control

format: `return (`*calculational_expression*`)`

purpose: (1) Terminate the current script, and return to the calling script at the next line below the `do` or `include` command which called the current script.

(2) Optionally set the calling script's local variable `exitstat` to the value resulting from evaluation of the *calculational_expression*.

notes: If no argument is given, the calling script's `exitstat` will be set to zero. A `return` executed from a script which was not called via `do` (i.e., was invoked directly as an invocation argument to OJ itself) will behave like `exit`, and the OJ process will have the exit status resulting from the evaluation of the optional expression.

The *calculational_expression* should always yield an integer value.

For information on calculational expressions, see Section 8.

input/output

seekfile

format: `seekfile` *file_handle offset place*

purpose: Position the file pointer to a location within (or at the end) of an open file. *offset* is the number of bytes before or after *place*. If you wish to offset before *place*, use a negative number for *offset*. The values of *place* are as follows:

0	beginning of file
1	current position
2	end of file

notes: Don't forget the $ before the *file_handle* variable. `qreturn` will be set to E_NO_ERR (0) if the seek is reasonable, or < 0 if it couldn't be done for some reason.

See also `openfile`, `closefile`, `readfile`, `writefile`, `fprintf`, and `tellfile`.

send

socket interaction

format: `send` *socket_handle* =
var_to_receive_name_of_matched_buffer string_1
(...string_n)

(NOTE: as with all commands, this one must be written all on one line in the script. If it won't fit, use the backslash line-continuation feature [see Section 6].)

purpose: Send a string to a `socket`ed process or device, await its reply, store its reply, and compare the reply to all associated match buffers.

notes: This command is central to socket interactions. The strings are concatenated and sent to the pseudo-socket (the device or process) specified in a preceding `socket`, which returned the *socket_handle* value. If the `send` was preceded at some point by a

```
socket sockhandle = /t1
```

and then the `send` is executed:

```
send $sockhandle matched "my string"
```

The concatenated string sent to the socket is available, after the `send`, as

```
$sendbuf[$sockhandle]
```

A device's reply is stored in system variable `$reply[$sockhandle]`. In the case of a process-type `socket`, the process's stdout output is in `$reply[$sockhandle]`, and its stderr output is in `$ereply[$sockhandle]`. If any of the match buffers set via one or more `set_mbf` commands are matched, the name of the buffer matched is written on *var_to_receive_name_of_matched_buffer*. If no match buffer is matched, or if no match buffers were set prior to the `send`, or if an `ignore` command is executed prior to the `send`, the variable named in the second token will contain nothing (a null string). A special string in the concatenated string to be `sent`, `$PASSWORD`, will be replaced by the string obtained by employing the (optionally interactive) `password` command.

You must specify a device or process on which to perform i/o operations prior to the first `send`, by using a `socket`.

This command sets `qreturn`; check for equality to E_NOSOCKET,

E_PSJUSTDIED and E_PSWENTAWAY. send is a "backslash-interpreting" command.

See also ignore, password, set_mbf, set_send_delay, set_waits, show_mbfs, socket, strip_msb_on(_off), and verbose_on(_off).

set

string manipulation

format: `set` *resultvar* = (*option*) *string_1* (...*string_n*)

options:

=U	All lowercase characters in the entire string will be converted to uppercase.
=L	All uppercase characters will be converted to lowercase.
=R	The content of all variables will undergo backslash interpretation (see Section 10) as well as all hard strings.
=	No option used; no conversions will be made.

purpose: Concatenate strings into a single variable (*resultvar*).

notes: May have 4 or more tokens. `set` is a "backslash-interpreting" command (see Section 10 and the `alphameric` command, which has the opposite effect). To set a variable to a null string, use

```
set var = ""
```

socket interaction

set_mbf

format 1: (For use if *type_of_match_to_make* is BEGIN, END, or EXACT:)

```
set_mbf socket_handle  name_of_buffer_to_set
    type_of_match_to_make = string_1  (...string_n)
```

(NOTE: as with all commands, this one must be written all on one line in the script. If it won't fit, use the backslash line-continuation feature [see Section 6].)

format 2: (For use if *type_of_match_to_make* is B&E:)

```
set_mbf socket_handle name_of_buffer_to_set
    B&E separator = string_1  (...string_n)
    separator string_o  (...string_p)
```

purpose: Set a match buffer to the string concatenated from the tokens following the = sign (except *separator*).

notes: OJ will strive to match all the match buffers that are set during the following `send`. The possible values of *type_of_match_to_make* are:

BEGIN	Not very useful; the match will be made if the socket's reply begins with the string. All the waits required by the previous `set_waits` will occur.
END	The match will be made if the socket's reply ends with the string. As soon as the match is made, the `send` is over; the waits will not occur, so the time required to run the script is minimized.
EXACT	Like END but the match pattern and the socket's reply must exactly match if the match is to be made.
B&E ("Begin and End")	The strings before the *separator* will be concatenated and placed in the match buffer given in the *name_of_buffer_to_set* token. The strings after the *separator* will be concatenated and placed in a separate buffer. The socket's reply will be matched if both the beginning and the end of it match the two corresponding buffers.

Affects only the very next `send` encountered; `send` resets all match buffers. A "match buffer" is really just an ordinary variable with a special connection to

the next `send`; the special connection is destroyed by the `send`, but the variable is not destroyed.

In the case of a process-type socket, all the match buffers will be compared with both `reply[`*socket_handle*`]` and `ereply[`*socket_handle*`]` (i.e. stdout and stderr).

`set_mbf` is a "backslash-interpreting" command.

external process control **set_prior**

format: `set_prior` *calc_expression_yielding_process_priority_value*

purpose: Set the processing priority of the OJ interpreter (and the default priority of all the processes it will fork later).

notes: Beware setting priority so low it never gets executed, or so high it stops all other processes. See the OS-9 manual regarding the setpr utility for more information. See also Section 8 of this manual.

setrand

random numbers

format: `setrand` *seed_string*

purpose: Set the seed from which all future random numbers (generated during the current invocation of the interpreter) will be generated, at least until the next `setrand`.

notes: This command affects the `randlist` command and the `rand()` function.

There is generally no reason to use this command except during debugging. When debugging, it is sometimes useful to set the seed so that the random numbers generated will always be the same. If there has been no `setrand`, the first time a `rand()` or a `randlist` is executed, the seed is set automatically to a value based on the current date and time. If you do not use `setrand`, and instead rely on the automatic seeding feature, it should not be possible to get the same automatic seed twice within the same 500,000-day period. A comforting thought!

The *seed_string* must be exactly four bytes long. It cannot consist entirely of 0 bytes (i.e., `"\x00\x00\x00\x00"`), however.

socket interaction

set_send_delay

format: `set_send_delay` *socket_handle integer_value*

purpose: Set the number of 256ths of a second to pause after sending each byte to the `socketed` device or process.

notes: The default setting is `0`. Typically you shouldn't need to use this command at all, but it is provided in order to accommodate certain devices that can't accept bytes at full speed.

Exactly 3 tokens. `send` does not reset this parameter. See also `send`.

set_waits

socket interaction

format: `set_waits` *socket_handle integer_value*

purpose: For all succeeding `send` commands, set the number of half-second wait intervals to await more output from the `socketed` device or process, after the last byte is received.

notes: Think of this as a "maximum pause with no response from the `socketed` device or process." If the output from the `socketed` device or process is matched by an EXACT, END, or B&E match buffer, these waits will not take place. If there is no match, or no match buffers have been set via one or more `set_mbf` commands, or an `ignore` command is in effect, these waits will all take place unconditionally. If no `set_waits` has been executed, the number of waits will be 10 (i.e. 5 seconds).

`send` does not reset this parameter.

See also `send`, `socket`, `set_mbf`, `ignore`, and `verbose_on`.

debugging

show_comlines_on (show_comlines_off)

format: `show_comlines_on`

or

`show_comlines_off`

purpose: For debugging. Causes OJ to output the command lines to the console (via stderr) as they are encountered, before they undergo any processing. Also shows whether the interpreter is currently searching for a particular label, and the current nesting level of `if`.

notes: Exactly 1 token. This is probably not the command you want. You probably want `echo_on` and/or `verbose_on`, or you want to use the debugger. See `echo_on`, `verbose_on`, and Section 12.

show_labels

debugging

format: `show_labels`

purpose: For debugging. Causes OJ to output all known statement labels, and their line and byte numbers within the current script, to the console via stderr.

notes: Exactly 1 token. Statement labels that have not yet been encountered will not appear in the list. This command is probably not what you are looking for, unless you are doing tricky stuff which uses the content of variables as labels.

socket interaction/ debugging

show_mbfs

format: `show_mbfs` *socket_handle*

purpose: For debugging. Displays the currently active match buffers on the console (via stderr).

notes: Remember that after a `send` all match buffers are cancelled. Therefore this command is useful only after one or more `set_mbf` commands, but before their associated `send`. See also `send` and `set_mbf`.

show_tokens_on (show_tokens_off)

debugging

format: `show_tokens_on`

or

`show_tokens_off`

purpose: For debugging. Causes OJ to output all tokens to the console via stderr immediately after tokenization and before variable substitution.

notes: Exactly 1 token. Makes an extremely verbose display. This command is probably not what you want, unless you are wondering how a command line is being tokenized. See `echo_on`.

debugging

show_vars

format: `show_vars`

purpose: For debugging. Causes OJ to output the names and contents of all extant variables to the console via stderr.

notes: Exactly 1 token. It's usually easier to use the `p` command in the interactive debugger (see Section 12), but `show_vars` gives you a quick and dirty way to display the entire variable inventory on the screen.

socket

socket interaction

format 1: `socket` *var_for_socket_handle* = *device_name*

format 2: `socket` *var_for_socket_handle* = *pipe_size_in_bytes*
process_name (*process_arg_1*... *process_arg_n*)

(NOTE: as with all commands, this one must be written all on one line in the script. If it won't fit, use the backslash line-continuation feature [see Section 6].)

format 3: `socket` *socket_handle*

purpose: Specifies the device (format 1) or invokes the process (format 2) with which to exchange data via the `send` command, or kills a process invoked by a previous `socket` (format 3). Formats 1 and 2 are used to set a variable to a socket handle value, which is required by all other socket interaction commands.

notes: Any number of `socket`ed processes and devices can be kept on hand simultaneously; each is independently addressable via its associated socket handle, via the `send` command. (See `send`.)

You must execute this command prior to using `send`. Typically, for interactions with a host computer system, you will use format 1 and specify `/t1` or another serial port device. For interactions with some local process, such as Kermit, simply specify the variable name to receive the socket handle, the pipe size in bytes (minimum size is 256 bytes), and `kermit`:

```
socket var_for_socket_handle = 256 kermit
```

In the case of a process, the process is invoked at the `socket` and it remains active until killed by a `socket` (format 3), or until it dies spontaneously, or OJ terminates. This command sets `qreturn`; be on the lookout for E_PSNOFORK and E_PSWENTAWAY values.

string manipulation

sort

format: `sort` *source_array* *sorted_array* `(reverse)`

purpose: Determine the lexicographical (or reverse lexicographical) order of the items in an array. To get reverse order, use the optional final token, `reverse`.

notes: *source_array* must be the source array name, which is a variable that must exist and must have as its value the number of elements in *source_array*[ ]. *sorted_array* is the name of the receiving array. This variable will be created, if necessary, as will all the elements of *sorted_array*[ ]. N.B.: to force the creation of the array as local variables, precede the `sort` with a

```
local sorted_array
```

The value contained in *source_array* will be copied into *sorted_array*. The content of each element of *sorted_array*[ ] will be the index number of the corresponding element in *source_array*[ ], not the value of the corresponding element in *source_array*[ ]. For example: (first we have to have have an array to sort)

```
set sourcearray[0] = my
set sourcearray[1] = dog
set sourcearray[2] = has
set sourcearray[3] = fleas
set sourcearray = 4      # length of the array
```

Then we sort it:

```
sort sourcearray targarray
```

`sourcearray` is not affected.
`targarray` has now been created, and it looks like this:

```
$targarray is "4";
$targarray[0] is "1"; so
   $sourcearray[$targarray[0]] is "dog"
$targarray[1] is "3"; so
   $sourcearray[$targarray[1]] is "fleas"
$targarray[2] is "2"; so
   $sourcearray[$targarray[2]] is "has"
$targarray[3] is "0"; so
   $sourcearray[$targarray[3]] is "my"
```

To display the items in sourcearray in alphabetical order, then:

```
loop 0end ; i = 0 ; i < $targarray; \
     i = i + 1
     writefile $stdout $i "  " $sourcearray[$i] \
          "  " $targarray[$i] \
          "  " $sourcearray[$targarray[$i]] \n
0end
```

The above loop's output is:

```
0  my   1  dog
1  dog   3  fleas
2  has   2  has
3  fleas  0  my
```

If the sort command had been:

```
sort sourcearray targarray reverse    # reverse
```

The output would have been:

```
0  my  0  my
1  dog  2  has
2  has  3  fleas
3  fleas  1  dog
```

user interaction

spilldev

format: `spilldev` *device_name*

purpose: Pick up all bytes waiting at the named SCF device, and discard them.

notes: Useful for clearing the standard input stream after displaying a dangerous prompt.

```
fprintf $stdout \
     "Press RETURN to launch nuclear weapons. \n"
spilldev /term    # no hangover RETURN key allowed to
                  #    start war.
stdin x
     # launch nuclear weapons here.
```

sprintf

string manipulation

format: `sprintf` *var_to_be_set control_string (string_1...string_n)*

purpose: Write a formatted string on a variable using the same control string and conversion conventions as the sprintf() function in the C language.

notes: Please refer to Kernighan and Ritchie's The C Programming Language or any other good book on C. `sprintf` is a "backslash-interpreting" command. See also `fprintf`.

user interaction **stdin**

format: `stdin` *var*

purpose: Pick up a newline-terminated line from the user (or stdin).

notes: Don't use a `$` unless you want the input to be placed in a variable whose name is the same as the content of *var*. `stdin` calls OS-9's readln() function.

strip_msb_on (strip_msb_off)

socket interactions

format: `strip_msb_on` *socket_handle*

or

`strip_msb_off` *socket_handle*

purpose: Cause the most significant bit (msb) of each incoming byte from a `socketed` device or process to be reset (or left unchanged).

notes: When chatting with a remote host computer, the most significant bit of each incoming byte is often used for parity checking. It is undesirable to allow the parity bit to come through, because when it is set, it renders the incoming byte uninterpretable as a character. For this reason, after `socketing` a device, the incoming data will have its most significant bit stripped unless a `strip_msb_off` command is issued. The default situation for `socketed` processes is the reverse: the most significant bit of each incoming byte will be left unchanged unless a `strip_msb_on` command is encountered.

string manipulation

substring

format: `substring` *var* =
string_from_which_substring_is_to_be_copied
starting_position (*length_of_substring_to_be_copied*)

(NOTE: as with all commands, this one must be written all on one line in the script. If it won't fit, use the backslash line-continuation feature [see Section 6].)

purpose: Put a copy of a portion of a string into another variable.

notes: To copy from the very beginning of the *string_from_which_substring_is_to_be_copied*, *starting_position* should be 0. If *length_of_substring_to_be_copied* is omitted, all of the remainder of the source string will be copied. BEWARE: the source string must be a single token. Therefore, if it is a hard string (i.e., it is not given as a *varname*), and it has whitespace in it, it must be surrounded with double quotation marks.

switch

flow of control

format: `switch` *end_label* *string_to_match_with_each_case*

purpose: Executes all the code after the first matching `case` until a subsequent `break` is encountered.

notes: This works pretty much like switch in the C language, except that each `case` is a string match rather than a value match. When the `switch` command is encountered, all the tokens after the *end_label* are concatenated into a single string; this single string will be compared with the string given in each `case`.

Remember: just as in C, once a `case` has been matched, all the code in all succeeding cases will be executed until there is a `break`. If no `case` is matched, the code after the `default`, if any, will be executed. Consult a book on the C language for details. See also `break`.

```
# example of a switch:
switch 100_end_switch $string_to_match
    case "hi mom"
          # execution resumes here if the content
          # of string_to_match is "hi mom"
        break
    case 23
    case $quodlibet
          # execution resumes here if either "23"
          # or the content of variable
          # quodlibet matches string_to_match
        break
    default
          # execution resumes here if
          # no case matches string_to_match
100_end_switch
```

debugging

sys_exec_off (sys_exec_on)

format: `sys_exec_off`

or

`sys_exec_on`

purpose: For debugging. Disable (or enable) invocation of programs called in a script.

notes: Useful in checking flow of control in scripts before allowing OJ actually to call any external programs. Best used with `echo_on`. In the absence of a `sys_exec_off`, external program invocation is enabled. `sys_exec_on` has no effect if an `--x` option was used at OJ invocation time (see Section 4). Exactly 1 token.

tellfile

input/output

format: `tellfile` *file_handle*

purpose: Find the current position of the file pointer within an open file. Sets `qreturn` to the byte number within the file, or < 0 if there was an error.

notes: Don't forget the $ before the file handle variable. By saving the the file pointer position in `fpvar`:

```
tellfile $fhandle
set fpvar = $qreturn
```

you can get back to the same position by:

```
seekfile $fhandle $fpvar 0
```

See also `seekfile`, `openfile`, `closefile`, `readfile`, `writefile`, and `fprintf`.

memory management

unset

format: `unset` (*option*) *varname_1* (*...varname_n*) (*option*)

options: -LOCAL
-GLOBAL
-LOCGLOB (both; default)

purpose: (1) Recover the memory allocated to one or more variables by destroying the variables altogether.

(2) Destroy a local variable which has the same name as a global variable, so as to regain access to the global variable.

notes: Using the -LOCAL option will destroy only the variables that exist in the local variable bank; using the -GLOBAL option will similarly restrict destruction to the global variable bank. Using no option (or the -LOCGLOB option) will destroy all matching variables in both the local and the global variable banks.

Each variable name can include wildcards. The ? wildcard can be any single character; the * wildcard represents any combination of 0 or more characters. To destroy all the variables, just use a *. You cannot destroy system variables.

This command sets `qreturn` to E_VNOTFOUND if any of the variables could not be found in the designated bank(s).

verbose_on (verbose_off)

socket interaction/ debugging

format: `verbose_on` *socket_handle*

or

`verbose_off` *socket_handle*

purpose: If a `verbose_on` has been executed, a running status display will be sent to the console via stderr during each `send`. The status display will tell what is being sent, what the `socket`ed process or device replies, how long a wait was required, etc.

notes:

When optimizing the script, `set_waits` should often be set to a number which is some what higher than the largest number of waits that allows for invoked sluggishness of the `socket`ed process to respond. The only way to find out how many waits are required is to observe the display generated by a `verbose_on` in effect. See Section 4.

memory management/ data storage

vrestore

format: `vrestore` *filename varname_1 (...varname_n)*

purpose: Restore from the file all of the variables whose names are given. The file must have been made by using the `vsave` command.

notes: Each variable name can include wildcards. The ? wildcard will match any single character; the * wildcard represents any combination of 0 or more characters. To restore all the variables in the file, just use a *. For an example, see `vsave`.

This command sets `qreturn`, which must be checked if you are at all interested in reliability. If `qreturn` is less than 0, the save was not done, except when `qreturn == E_VNOTFOUND`, in which case at least one variable was not restored because it could not be found, but the rest (if any) were restored.

vsave

memory management/ data storage

format: `vsave` *filename* (*option*) *varname_1* (*...varname_n*)

purpose: Save in the file all of the variables whose names are given, for later recovery via the `vrestore` command.

options:

`-NEW`	The file, if it exists, will be completely overwritten.
`-UPDATE` (default)	Only the named variables will be updated (and created, if necessary) within the file. This method is far more time-consuming, but often worthwhile.

notes: Each variable name can include wildcards. The ? wildcard can be any single character; the * wildcard represents any combination of 0 or more characters. To save all the variables, just use a *. System variables such as `argc` and `qreturn` are never saved. If you want to save their values, simply copy them to other variables and save those variables.

Local variables will be saved as local variables, and they will be restored as local variables by `vrestore`. Similarly, global variables will be saved and restored as global variables.

This command sets `qreturn`, which must be checked if you are at all interested in reliability. If `qreturn` is less than 0, the save was not done, except when `qreturn == E_VNOTFOUND`, in which case at least one variable was not saved because it could not be found, but the rest (if any) were saved.

Example:

```
vsave myvars.vs -update scores[*] donecount \
   exam* *x*
if (qreturn < 0)
   fprintf $stderr "vsave failed; %s\n" \
      $e_msg[$qreturn]
   exit 1
endif
```

In the example, a variable storage file whose name is "myvars.vs" will be created if it doesn't exist, or updated if it does exist. All currently extant variables which meet the following criteria will have their values in the file updated (if they

already exist) or they will be entered in the file for the first time with their current values):

(1) all variables in array `scores[...]`
(However, `scores[whatever][whatever_else]` will not be saved. To save it, you might use `scores*`.)

(2) variable `donecount`

(3) all variables whose names begin with `exam`

(4) all variables whose names contain at least one `x`.

wait

miscellaneous

format: `wait` *number_of_half_seconds* `(-verbose)`

purpose: Unconditionally stop further execution of the script for the number of half seconds given in the tag.

notes: Sometimes useful in auto-login scripts on systems which ignore input for some period of time after displaying the "login:" prompt. The `-verbose` option causes a verbose-type display to appear on the console, showing the current wait number and the total waits. If the `--v` invocation option was used, it has the effect of adding the `-verbose` option to all `wait` commands.

input/output

writefile (-writefile)

format: `writefile` *file_handle string_1 (... string_n)*

or

`-writefile` *file_handle string_1 (... string_n)*

purpose: Write some bytes on a file (or to `$stdout` or `$stderr`).

notes: `qreturn` will be set to a negative value if an error occurred, or it will be set to a positive value equal to the number of bytes written. All the string tokens will be concatenated and written to the file.

Don't forget the `$` before the file handle variable. To output text to the screen, use `writefile $stdout` or `writefile $stderr`.

`writefile` uses fputc(), which means that when the output is to an SCF device (such as `/t1`) rather than to a file, the translations in effect for that device will be done. These translations may include the expansion of tab characters into some number of spaces, and the addition of a linefeed character to each carriage return character. If you do not wish these translations to take place, use the `-writefile` rather than the `writefile` form of the command. `-writefile` calls the low-level write( ) function rather than fputc( ); write( ) does not perform the translation. (For more information about translation of characters, consult the documentation on the x mode program in your OS-9 manual.)

`writefile` is a "backslash-interpreting" command.

See also `openfile`, `fprintf`, `closefile`, `readfile`, `seekfile`, and `tellfile`. It's often handier to use `fprintf`.

WARNING: Once a device or file has been opened, use either `writefile` or `-writefile`, but do not use both. Also, if you use `-writefile`, the only other i/o command you may use with its file handle is `closefile`. Do not attempt to use `fprintf`, `readfile`, `seekfile` or `tellfile`— the results will be unpredictable.

This page deliberately left blank.

The debugger is built into the OJ interpreter. It allows the programmer to control and interrupt execution of scripts at arbitrary times, and, while execution is interrupted, to examine the contents of any or all variables. To be in debug mode, use the `--d` option when invoking OJ. You will then see the first executable line of the script, followed by the prompt:

```
OJdb:
```

The line of your script which is displayed immediately above the `OJdb:` prompt is always the line which is about to be executed.

At this prompt you may type any of the following single-letter commands (with their argument[s] as appropriate), and then press the RETURN key. Each discussion of a single-letter command begins with a {mnemonic} word to help you remember what the letter means. Some effort has been made to make operation of the debugger similar to that of the OS-9 C language source debugger, although there are (of course) numerous differences.

?	{Help}	Lists information about the debugger on the screen.
$	{Shell}	Allows temporary escape to an OS-9 shell prompt. Logging out of the shell will return you to where you were, at an `OJdb:` prompt. Upon returning, you may wish to use the `i` and/or `f` commands to see where you are. WARNING: It is extremely dangerous to edit any script which has been left open by the OJ interpreter you are temporarily escaping. When you come back and restart execution, literally anything can happen, including the destruction of valuable files. Be safe: fully exit OJ before editing any OJ scripts; do not use this shell escape for editing scripts.
b	{Breakpoint}	Displays the breakpoints currently in effect. Breakpoints are places in the script, described as "*scriptname\line_number*" (note the backslash between the scriptname and the line number), where you wish to halt execution and get an `OJdb:` prompt. If the line number is 0, that means "halt at the first executable line."

`b` *scriptname* {Breakpoint}

Sets a breakpoint at the first executable line of script *scriptname*. One arrives at such a point immediately after any `do` or `include` command, at the first executable line of the `included` or `done` script. When the breakpoint is encountered, execution halts and an `OJdb:` prompt is displayed. The actual breakpoint line number will be 0, which means "halt at the first executable line".

`b` *scriptname\line_number* {Breakpoint}

Sets a breakpoint at the line number given after the backslash. The line number must have an executable command on it, or the debugger will not "see" it, but will pass right over it without stopping. Execution will halt on that line number if and only if that line is supposed to be executed. For instance, if you put a breakpoint at a line which is never executed because it is in an `if` region whose condition is never satisfied, OJ will not halt there.

`b` *line_number* {Breakpoint}

Just like the above command, except that it stops at the line number within the current script.

`c` *calculational_expression* {Calc}

Type anything here that could appear after the = token in a `calc` (or as the tokens following an `if`), and it will be evaluated exactly the same way that it would be if it occurred in a command in the script when the `OJdb:` prompt appeared. The evaluation, which is always a real number, will be displayed. Do not use an = sign. No OJ variable will be set by using this command (for that, use the `sl` or `sg` commands).

`e1` {Echo-true} Turns on echo mode. This has the identical effect of an `echo_on` command or an invocation of OJ with a `--e` argument.

`e0` {Echo-false} Turns off echo mode. This has the identical effect of an `echo_off` command or an invocation of OJ without a `--e` argument.

`f` {Frame} Displays the current "frame" or context. The current script name and line number are displayed first, followed by all script names and line

numbers containing `do` or `include` commands that led to your current location (the "do" stack).

`g` {Go}

Executes the script until a breakpoint is encountered or OJ terminates. See the `b` command for information on setting breakpoints. See also the `r` debugger command.

No echoing of lines will be done unless an `echo_on` command is/was encountered in the script, the `--e` option was used at OJ invocation time, or the `el` command was issued at an `OJdb:` prompt.

`g` *scriptname* {Go}

Execute script until *scriptname* is invoked by means of a `do` or `include` command.

`g` *scriptname**line_number* {Go}

Execute the script until line *line_number* is encountered in script *scriptname*. The specified line must have a command on it which is supposed to be executed (i.e., which is not within an unsatisfied `if` region, etc.), or the debugger will not "see" it, but will pass right over it without stopping.

`g` *line_number* {Go}

Execute script until line *line_number* is encountered in the current script. The line number must have a command on it which is supposed to be executed (i.e., which is not within an unsatisfied `if` region, etc.), or the debugger will not "see" it, but will pass right over it without stopping.

NOTE: All of the above argumented `g` ("go until...") commands actually set a temporary breakpoint which goes away as soon as it is first encountered. In other words, the breakpoint set with an argumented `g` command only works once. To set a breakpoint that is not unset when it is encountered, use the `b` command.

`i` {Info}

Redisplays the line you are about to execute.

`k b` *breakpoint_number* {Kill}

Kills a breakpoint previously created by the `b` command. First, do a `b` command with no argument; this will show all the breakpoints. The breakpoints are identified as b0, b1, b2, etc. Decide which one you want to kill, and enter its name (e.g., `b0`) as the argument of a `k` command:

```
OJdb: k b0
```

`k *` {Kill} Kills all breakpoints.

`l` {List} Displays a numbered-line listing of the current script, starting at the current line in that script, one screen at a time. If all the script file has not yet been shown, the prompt will be:

```
OJdb [LIST]:
```

In order to see the next screenful of text, press RETURN.

`l` *line_number* {List}

Like `l`, above, except the listing will begin with the line number given in the argument.

`l` *text_file_name* {List}

Like `l`, above, except that you will be looking at any arbitrary text file whose name you give in the argument. This of course includes scripts other than the one you are presently executing.

`l` *text_file_name**line_number* {List}

Displays the named text file, starting at the specified line number. Be sure to use a backslash between the two arguments.

`n` {Next} Executes the script line already displayed above the prompt, shows the next line to be executed, and gives another `OJdb:` prompt. When you press `n` the first time, the `OJdb:` prompt becomes `OJdb[NEXT]:`. This means that you don't have to press `n` followed by RETURN to repeat the `n` command; instead, just press RETURN all by itself.

`n` *number* {Next} Executes the specified number of lines in the script, echoing each of them before execution; it's equivalent to issuing the `n` command that number of times.

`p` *variable_name* {Print}

Displays the contents of the variable named in the argument. You may use wildcard characters (`?`, `*`) in the variable names to see several variables at once. An asterisk all by itself will display all variables.

`q` {Quit} Aborts execution of OJ and returns you to your original shell prompt.

`r` {Return} Executes the remainder of the current script, halting automatically when you return to the calling script.

`sl` *var_to_set* = *strings...* {Set Local}

Set a local variable to some string value. See `sg` below.

`sg` *var_to_set* = *strings...* {Set Global}

Set a global variable to some string value. The `sl` and `sg` commands behave identically to the `set` command, except that you can be specific about whether you wish to set a local or a global variable. WARNING: These debugging commands have no impact on the script itself except during the current debugging session; the script itself still has to be fixed.

This page deliberately left blank.

These are the names of the pseudo-variables containing the values returnable by qreturn, together with their actual numeric values and the strings found in the e_msg[$qreturn] array. WARNING: Don't use the values; use the names instead. The values may change, but the names won't.

$E_NO_ERR	0	no reportable error has occurred
$E_FOPEN	-1	file could not be opened via fopen()
$E_NONINT	-2	an integer value was required; this wasn't an integer
$E_BADPLC	-3	place value for seek() was not 0, 1, or 2
$E_BADSEEK	-4	fseek() failed
$E_BADTELL	-5	ftell() failed
$E_PSWENTAWAY	-6	process already was dead
$E_PSJUSTDIED	-7	process just died; normal after a regular OS-9 invocation; if a socketed process, another socket is now required
$E_PSNOFORK	-8	process could not be forked at all
$E_INVSOCKH	-9	invalid socket handle
$E_INVFH	-10	invalid file handle

$E_BADCHD	-11	unsuccessful attempt to change data directory
$E_BADCHX	-12	unsuccessful attempt to change execution directory
$E_CVCHTOOMANY	-13	more %'s in control string than arguments
$E_BADCONVSTR	-14	conversion string beginning with % is ill-formed
$E_CVCHTOOFEW	-15	fewer %'s in control string than arguments
$E_RFEOF	-16	file is exhausted -- no more data available
$E_NOTVFILE	-17	file exists but is not a variable storage file
$E_CORRVFILE	-18	existing variable storage file is corrupt
$E_VF_NO_WRITE	-19	could not write variable storage file
$E_VNOTFOUND	-20	while saving, restoring, or unsetting variables, at least one requested variable did not exist

INVOCATION OPTIONS

FUNCTIONS USED IN CALCULATIONAL EXPRESSIONS

OJ COMMANDS

CALCULATION

DATA STORAGE

DEBUGGING

EXTERNAL PROCESS CONTROL

FLOW OF CONTROL

INPUT/OUTPUT

MEMORY MANAGEMENT

MISCELLANEOUS

RANDOM NUMBERS

SOCKET INTERACTION

STRING MANIPULATION

TIME MEASUREMENT

USER INTERACTION

This page deliberately left blank.

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U